Racket Functions

Functions: most important building block in Racket (and 251)
• Functions/procedures/methods/subroutines abstract over computations
• Like Java methods, Python functions have arguments and result
• But no classes, this, return, etc.

Examples:

```
(define dbl (lambda (x) (* x 2)))
(define quad (lambda (x) (dbl (dbl x))))
(define avg (lambda (a b) (/ (+ a b) 2)))
(define sqr (lambda (n) (* n n)))
(define n 10)
(define small? (lambda (num) (<= num n)))
```

Function calls (applications)

To use a function, you call it on arguments (apply it to arguments).
E.g. in Racket: (dbl 3), (avg 8 12), (small? 17)

Syntax: (E0 E1 ... En)
• A function call expression has no keyword. A function call because it’s the only parenthesized expression that doesn’t begin with a keyword.
• E0: any expression, known as the rator of the function call (i.e., the function position).
• E1 ... En: any expressions, known as the rands of the function call (i.e., the argument positions).

Evaluation rule:
1. Evaluate E0 ... En in the current environment to values V0 ... Vn.
2. If V0 is not a lambda expression, raise an error.
3. If V0 is a lambda expression, returned the result of applying it to the argument values V1 ... Vn (see following slides).
Function application

What does it mean to apply a function value (lambda expression) to argument values? E.g.

```lisp
((lambda (x) (* x 2)) 3)
((lambda (a b) (/ (+ a b) 2)) 8 12)
```

We will explain function application using two models:

1. The **substitution model**: substitute the argument values for the parameter names in the function body.

2. The **environment model**: extend the environment of the function with bindings of the parameter names to the argument values.

Substitution notation

We will use the notation

\[ E[V_1, ..., V_n/Id_1, ..., Id_n] \]

to indicate the expression that results from substituting the values \( V_1, ..., V_n \) for the identifiers \( Id_1, ..., Id_n \) in the expression \( E \).

For example:

- \((* x 2)[3/x]\) stands for \((* 3 2)\)
- \(\(/ (+ a b) 2)[8,12/a,b]\) stands for \(\(/ (+ 8 12) 2)\)
- \((if (< x z) (+ (* x x) (* y y)) (/ x y))[3,4/x,y]\) stands for \((if (< 3 z) (+ (* 3 3) (* 4 4)) (/ 3 4))\)

It turns out that there are some very tricky aspects to doing substitution correctly. We'll talk about these when we encounter them.

Function application: substitution model

Example 1:

```lisp
((lambda (x) (* x 2)) 3)
```
Substitute 3 for \( x \) in \((* x 2)\) and evaluate the result:

\((* 3 2) \downarrow 6\) (environment doesn't matter in this case)

Example 2:

```lisp
((lambda (a b) (/ (+ a b) 2)) 8 12)
```
Substitute 3 for \( x \) in \((* x 2)\) and evaluate the result:

\((/ (+ 8 12) 2) \downarrow 10\) (environment doesn't matter in this case)

Big step function call rule: substitution model

\[
\begin{align*}
E_0 \uparrow_{env} & \quad \text{(lambda} \ (Id_1 \ ... \ Id_n) \ E_{body}) \\
E_1 \uparrow_{env} & \quad V_1 \\
& \quad \vdots \\
E_n \uparrow_{env} & \quad V_n \\
E_{body}[V_1 \ ... \ V_n/Id_1 \ ... \ Id_n] \uparrow_{env} & \quad V_{body} \quad \text{(function call)} \\
(E_0 \ E_1 \ ... \ E_n) \uparrow_{env} & \quad V_{body}
\end{align*}
\]

Note: no need for function application frames like those you’ve seen in Python, Java, C, ...
Substitution model derivation

Suppose $\text{env}2 = \text{dbl} \rightarrow \lambda (x) (* \, x \, 2)$, 
$\text{quad} \rightarrow \lambda (x) (\text{dbl} \, (\text{dbl} \, x))$

\[
\text{quad} \downarrow \text{env}2 \rightarrow (\lambda (x) (\text{dbl} \, (\text{dbl} \, x)))
\]
\[
3 \downarrow \text{env}2 \rightarrow 3
\]
\[
\text{dbl} \downarrow \text{env}2 \rightarrow (\lambda (x) (* \, x \, 2))
\]
\[
\text{dbl} \downarrow \text{env}2 \rightarrow (\lambda (x) (* \, x \, 2))
\]
\[
3 \downarrow \text{env}2 \rightarrow 3
\]
\[
(* \, 3 \, 2) \downarrow \text{env}2 \rightarrow 6 \text{ (multiplication rule, subparts omitted)}
\]
\[
\text{dbl} \, 3 \downarrow \text{env}2 \rightarrow 6
\]
\[
(* \, 6 \, 2) \downarrow \text{env}2 \rightarrow 12 \text{ (multiplication rule, subparts omitted)}
\]
\[
\text{quad} \, \text{dbl} \, 3 \downarrow \text{env}2 \rightarrow 12 \text{ (function call)}
\]
\[
(\text{quad} \, 3) \downarrow \text{env}2 \rightarrow 12
\]

Substitution model derivation: your turn

Suppose $\text{env}3 = n \rightarrow 10$, 
small? $\rightarrow \lambda \,(\text{num}) \, (\leq \, \text{num} \, n)$ 
sqr $\rightarrow \lambda \,(\text{num}) \, (* \, n \, n)$

Give an evaluation derivation for $(\text{small?} \, (\text{sqr} \, n)) \downarrow \text{env}3$

Stepping back: name issues

Do the particular choices of function parameter names matter?

Is there any confusion caused by the fact that $\text{dbl}$ and $\text{quad}$ both use $x$ as a parameter?

Are there any parameter names that we can’t change $x$ to in $\text{quad}$?

In $(\text{small?} \, (\text{sqr} \, n))$, is there any confusion between the global parameter name $n$ and parameter $n$ in $\text{sqr}$?

Is there any parameter name we can’t use instead of $\text{num}$ in $\text{small}$?

Small-step function call rule: substitution model

\[
(\lambda (\text{id}1 \ldots \text{id}n) \text{Ebody}) \, V1 \ldots Vn \rightarrow \text{Ebody}[V1 \ldots Vn/\text{id}1 \ldots \text{id}n] \text{ [function call]}
\]

Note: could extend this with notion of “current environment”
Small-step semantics: function example

\[
\begin{align*}
\text{quad} 3 \\
\Rightarrow & \quad ((\lambda (x) (\text{dbl} (\text{dbl} x))) 3) \\
\Rightarrow & \quad (\text{dbl} (\text{dbl} 3)) \\
\Rightarrow & \quad ((\lambda (x) (* x 2)) (\text{dbl} 3)) \\
\Rightarrow & \quad ((\lambda (x) (* x 2)) ((\lambda (x) (* x 2)) 3)) \\
\Rightarrow & \quad ((\lambda (x) (* x 2)) (* 3 2)) \\
\Rightarrow & \quad ((\lambda (x) (* x 2)) 6) \\
\Rightarrow & \quad (* 6 2) \\
\Rightarrow & \quad 12
\end{align*}
\]

Evaluation Contexts

Although we will not do so here, it is possible to formalize exactly how to find the next redex in an expression using so-called evaluation contexts.

For example, in Racket, we never try to reduce an expression within the body of a lambda.

\[
((\lambda (x) (+ (\ast 4 5) x)) (+ 1 2))
\]

We’ll see later in the course that other choices are possible (and sensible).

Small-step semantics: your turn

Use small-step semantics to evaluate \((\text{small? } (\text{sqr} n))\)

Assume this is evaluated with respect to the same global environment used earlier.

Recursion

Recursion works as expected in Racket using the substitution model (both in big-step and small-step semantics).

There is no need for any special rules involving recursion!

The existing rules for definitions, functions, and conditionals explain everything.

\[
\begin{align*}
\text{(define} \text{pow} \\
\text{(lambda} (\text{base} \text{ exp}) \\
\text{(if} (= \text{exp} 0) \\
1 \\
(* \text{base} (\text{pow} \text{ base} (- \text{exp} 1)))))
\end{align*}
\]

What is the value of \((\text{pow} 5 2)\)?
Recursion: your turn

Define and test the following recursive functions in Racket:

\( \text{fact } n \): Return the factorial of the nonnegative integer \( n \)

\( \text{fib } n \): Return the \( n \)th Fibonacci number

\( \text{sum-between } lo \text{ hi} \): Return the sum of the integers between integers \( lo \) and \( hi \) (inclusive) [You will do this in PS2 Problem 5]

Syntactic sugar: function definitions

**Syntactic sugar**: simpler syntax for common pattern.

- Implemented via textual translation to existing features.
- *i.e., not a new feature.*

Example: Alternative function definition syntax in Racket:

\[
\begin{align*}
\text{(define (} \text{IdFunName} \text{ Id1 ... Idn) Ebody)} \\
\text{desugars to} \\
\text{(define IdFunName (lambda (Id1 ... Idn) Ebody))}
\end{align*}
\]

- \( \text{(define (dbl x) (* x 2))} \)
- \( \text{(define (quad x) (dbl (dbl x))} \)
- \( \text{(define (pow base exp)} \)
  \text{if (< exp 1)} \)
  \text{1} \\
  \text{(* base (pow base (- exp 1))))} \)

Racket Operators are Actually Functions!

Surprise! In Racket, operations like \( (+ e1 e2) \), \( (< e1 e2) \) and \( (not e) \) are really just function calls!

There is an initial top-level environment that contains bindings for built-in functions like:

- \( + \) → *addition function*,
- \( - \) → *subtraction function*,
- \( * \) → *multiplication function*,
- \( < \) → *less-than function*,
- \( \text{not} \) → *boolean negation function*,
- ...

(where some built-in functions can do special primitive things that regular users normally can’t do --- e.g. add two numbers)

Summary So Far

Racket declarations:

- definitions: (define \text{Id E})

Racket expressions:

- conditionals: (if Etest Ethen Eelse)
- function values: (lambda (Id1 ... Idn) Ebody)
- Function calls: (Erator Erand1 ... Erandn)
  
  Note: arithmetic and relation operations are just function calls

What about?

- Assignment? Don’t need it!
- Loops? Don’t need them! Use tail recursion, coming soon.
- Data structures? Glue together two values with \text{cons} (next time)